Approximate Transformations as Mutation Operators

ICST'18 – Västerås, Sweden
April 12th, 2018

Farah Hariri†, August Shi†, Owolabi Legunsen†
Milos Gligoric‡, Sarfraz Khurshid‡, Sasa Misailovic†

† ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

‡ THE UNIVERSITY OF TEXAS AT AUSTIN

NSF
CCF- 1409423, CCF-1421503,
CCF-1566363, CCF-1629431,
CCF- 1652517, CCF-1703637,
and CCF-1704790

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 1
```

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 2
```

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 2
```

Approximate Computing

- **Goal:** Technique that trades accuracy for performance

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 2
```

Approximate Computing

- **Goal:** Technique that trades accuracy for performance
- **How:** Introduces small syntactic changes to the program
- **Example:**

```
for (i=0; i<n; i=i+1)
```

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 2
```

Approximate Computing

- **Goal:** Technique that trades accuracy for performance
- **How:** Introduces small syntactic changes to the program
- **Example:**

```
for (i=0; i<n; i=i+2)
```

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 2
```

Approximate Computing

- **Goal:** Technique that trades accuracy for performance
- **How:** Introduces small syntactic changes to the program
- **Example:**

```
for (i=0; i<n; i=i+2)
```

Motivation

Mutation Testing

- **Goal:** Technique that evaluates the quality of test suites
- **How:** Introduces small syntactic change to the program
- **Example:**

```
x = x + 2
```

Approximate Computing

- **Goal:** Technique that trades accuracy for performance
- **How:** Introduces small syntactic changes to the program
- **Example:**

```
for (i=0; i<n; i=i+2)
```

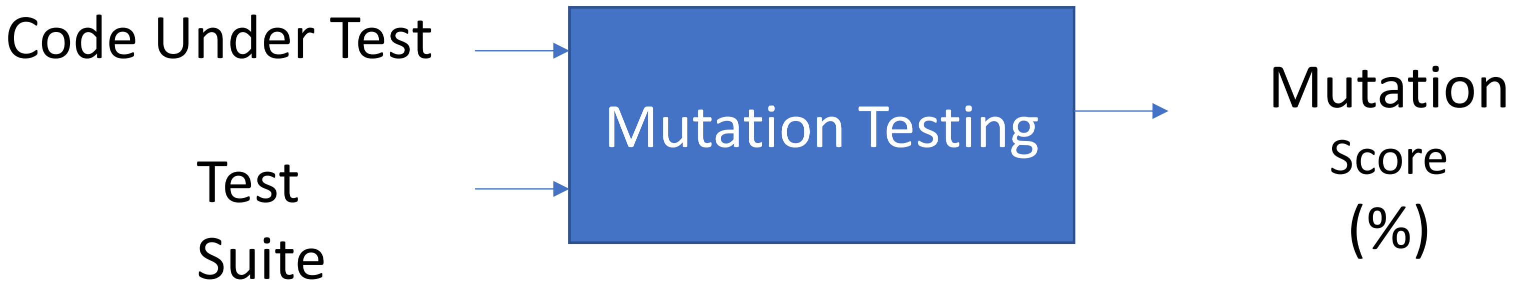
Vision

- To provide insights, mutation testing needs:
  - Varied mutation operators
  - High quality mutation operators

- We propose Approximate Transformations (ATs) as a new class of mutation operators leading to different program behaviors

Background: Mutation Testing

- Definition: Mutation testing is a technique for evaluating the quality of test suites



Code Under Test

Test Suite

Mutation Testing

Mutation Score (%)

STEPS OF MUTATION TESTING

01

MUTANT GENERATION

02

RUNNING THE TEST SUITE TO DETERMINE KILLED MUTANTS

STEPS OF MUTATION TESTING

01

MUTANT GENERATION

02

RUNNING THE TEST SUITE TO DETERMINE KILLED MUTANTS

Vectorz Mutation Example

```
public void swapRows(int i, int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
} }
```

Vectorz Mutation Example

```
public void swapRows(int i, int j) {
  if (i == j) return;
  int a = i * cols;
  int b = j / cols;
  int cc = columnCount();
  for (int k = 0; k < cc; k++) {
    int i1 = a + k;
    int i2 = b + k;
    double t = data[i1];
    data[i1] = data[i2];
    data[i2] = t;
 } }
```

STEPS OF MUTATION TESTING

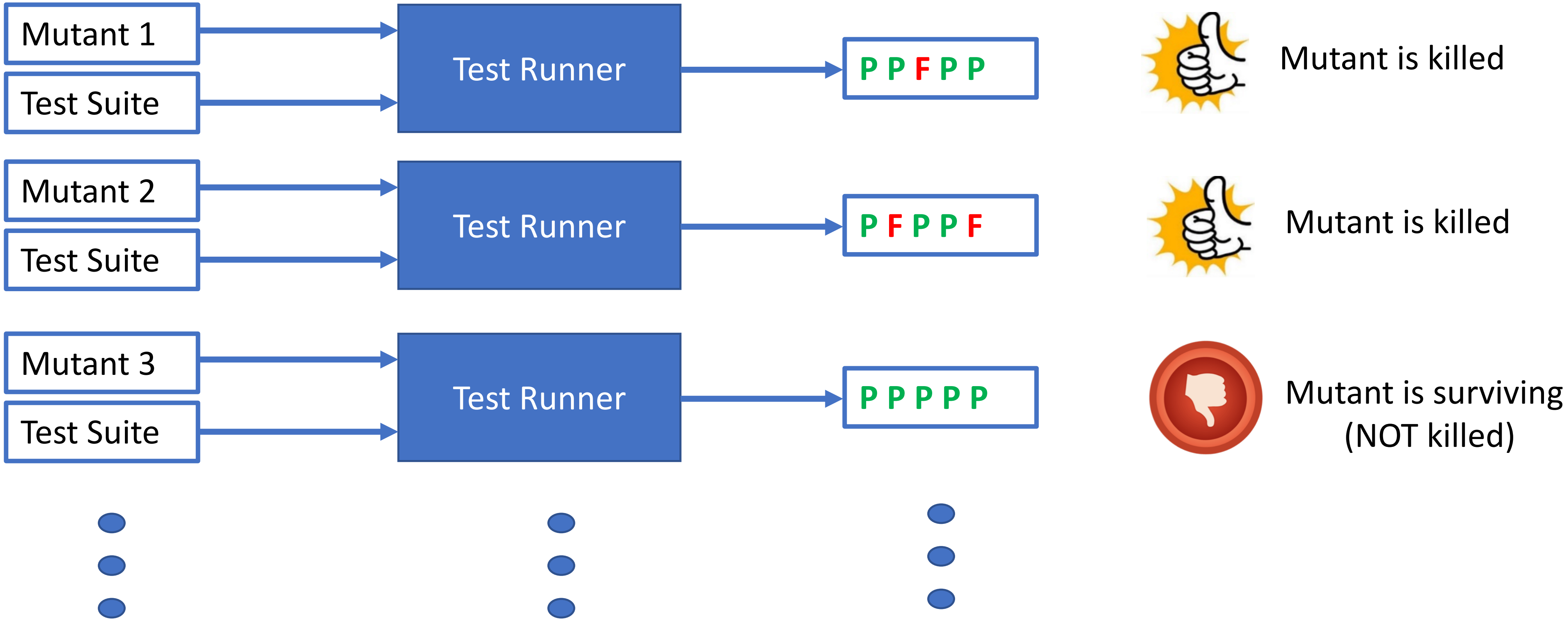
01

MUTANT GENERATION

02

RUNNING THE TEST SUITE TO DETERMINE KILLED MUTANTS

Running The Test Suite Outcomes



Mutant 1

Test Suite

Test Runner

P P F P P

Mutant is killed

Mutant 2

Test Suite

Test Runner

P F P P F

Mutant is killed

Mutant 3

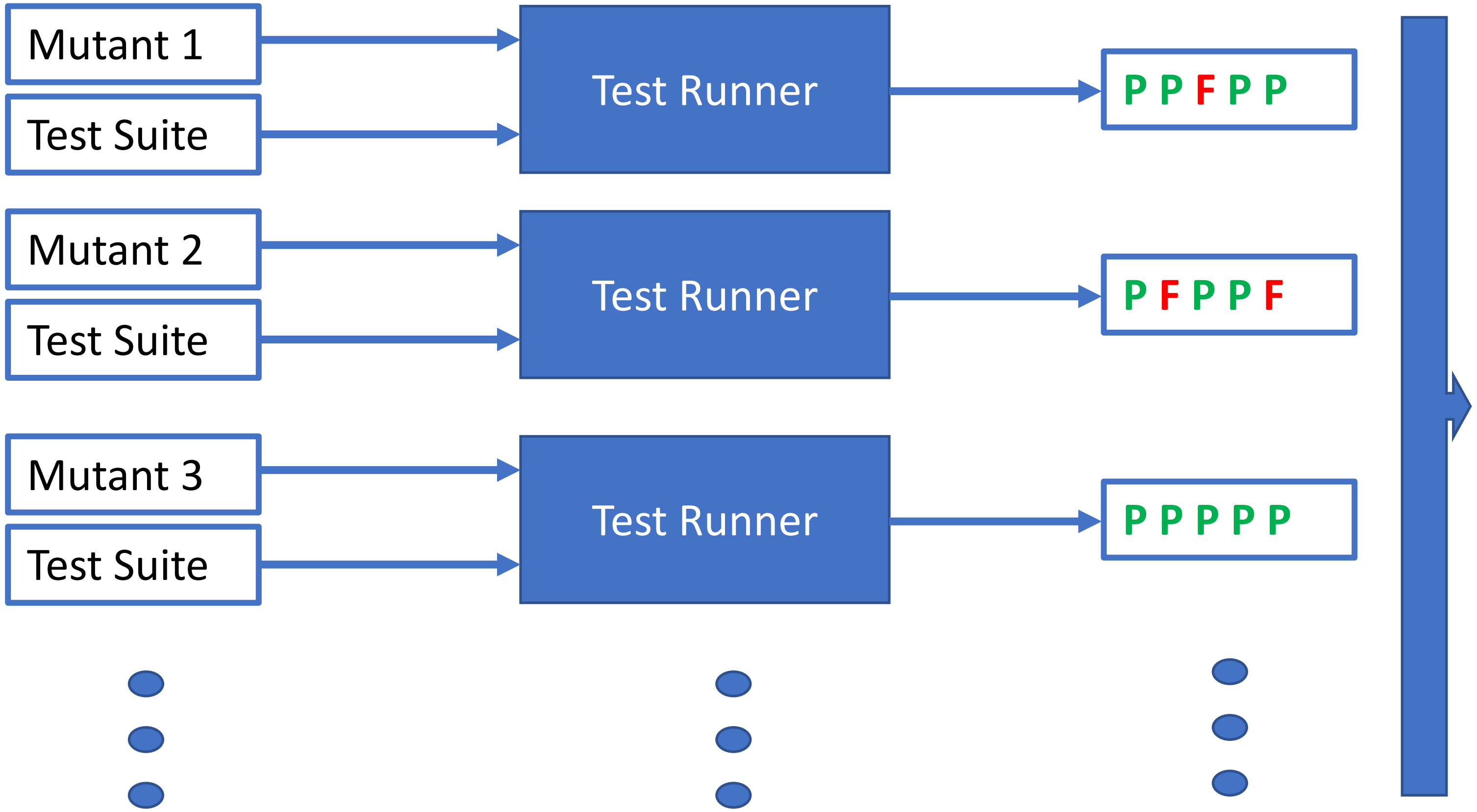
Test Suite

Test Runner

P P P P P

Mutant is surviving (NOT killed)

Mutation Score Computation



$$mutation\ score(\%) = \frac{killed\ mutants}{all\ generated\ mutants}$$

Vectorz Killed Mutant Example

```
public void swapRows(int i, int j)
{
 if (i == j) return;
 int a = i * cols;
 int b = j / cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
} }
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2)) return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

Vectorz Killed Mutant Example

```
public void swapRows(int i, int j)
{
 if (i == j) return;
 int a = i * cols;
 int b = j / cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 } }
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2)) return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

Mutant is killed

Background: Approximate Transformations

- Semantic changing transformations
- Trade accuracy for performance
- Big popularity in research recently
  - Machine learning, multimedia, data mining, …
- Example Transformations:
  - Integer to short precision degradation
  - Loop perforation

Vectorz Precision Degradation Example

```
public void swapRows(int i, int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
}}
```

Vectorz Precision Degradation Example

```
public void swapRows(int i, int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = (short) (a + k);
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 } }
```

Integer to short precision degradation

Vectorz Precision Degradation Example

```
public void swapRows(int i, int j)
{
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = (short) (a + k);
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 }}
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2)) return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

Mutant is NOT killed

Vectorz Loop Perforation Example

```
public void swapRows(int i, int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
} }
```

Vectorz Loop Perforation Example

```
public void swapRows(int i, int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k+=2) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
} }
```

Loop Perforation

Vectorz Loop Perforation Example

```
public void swapRows(int i, int j)
{
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k+=2) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 } }
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2)) return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

Vectorz Loop Perforation Example

```
public void swapRows(int i, int j)
{
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k+=2) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 }}
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2)) return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

👎 Mutant is NOT killed

Research questions

- **RQ1** How effective are ATs as mutation operators, compared to conventional mutation operators?
- **RQ2** What code patterns do ATs as mutation operators reveal?
- **RQ3** How can ATs as mutation operators help software testing practice?

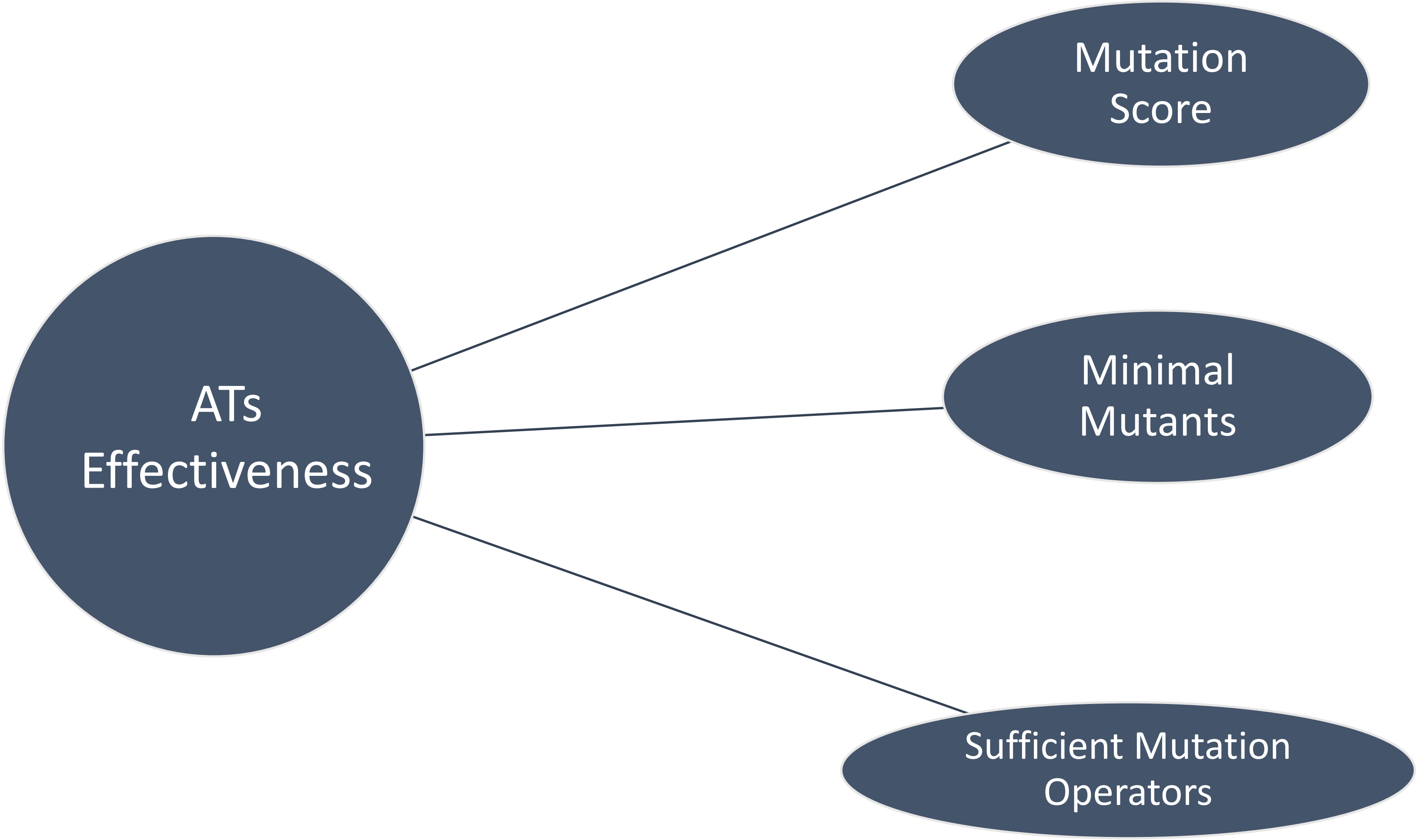
Experimental Setup

- We perform our study on 3 ATs
  - Loop Perforation (LPM)
  - Double To Float Precision Degradation (DTF)
  - Integer To Short Precision Degradation (ITS)
- Implement them as an extension to the PIT framework
- We compare against 14 PIT operators on 9 Open Source Java projects

RQ1:

How effective are ATs as mutation operators, compared to conventional mutation operators?

Quantitative Analysis



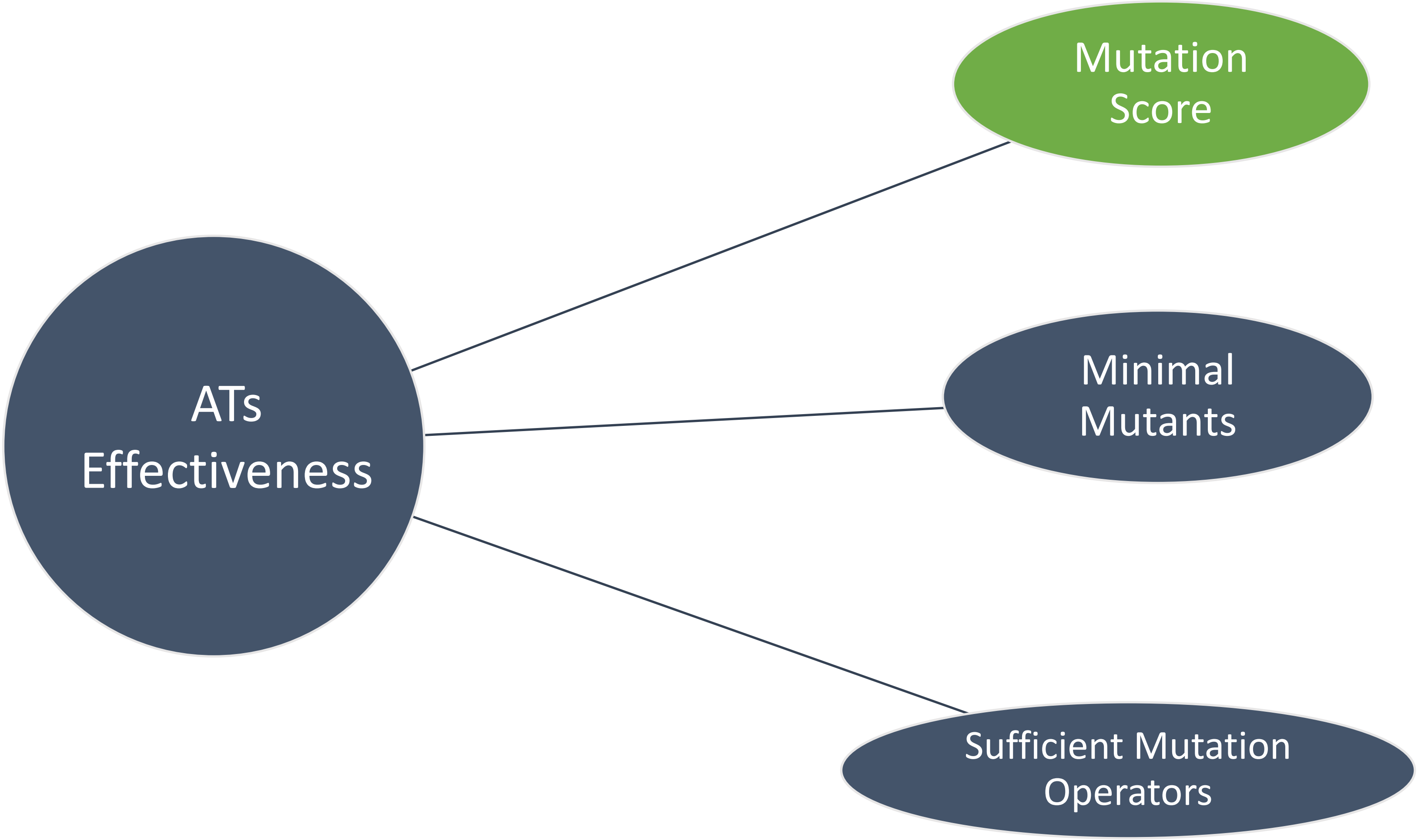
ATs Effectiveness

Mutation Score

Minimal Mutants

Sufficient Mutation Operators

Quantitative Analysis



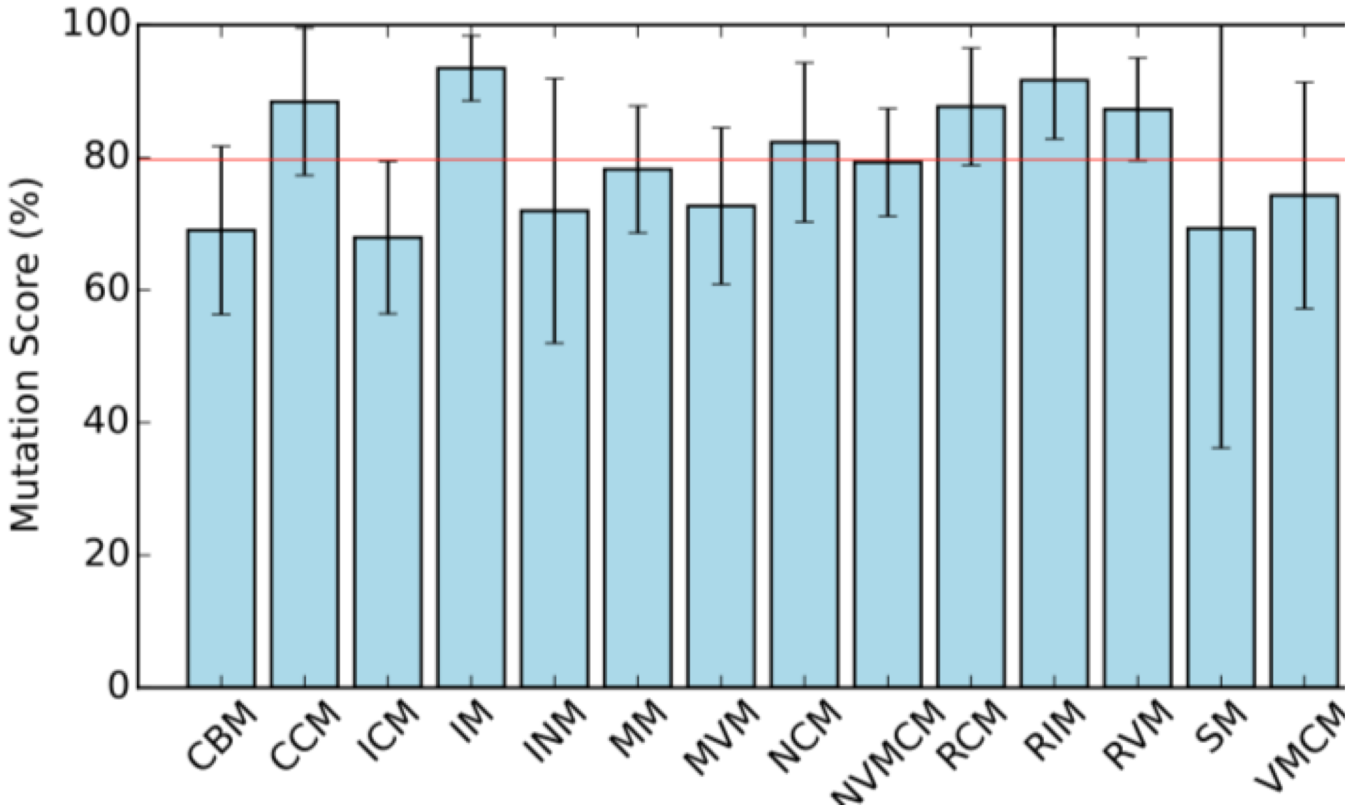
ATs Effectiveness

Mutation Score

Minimal Mutants

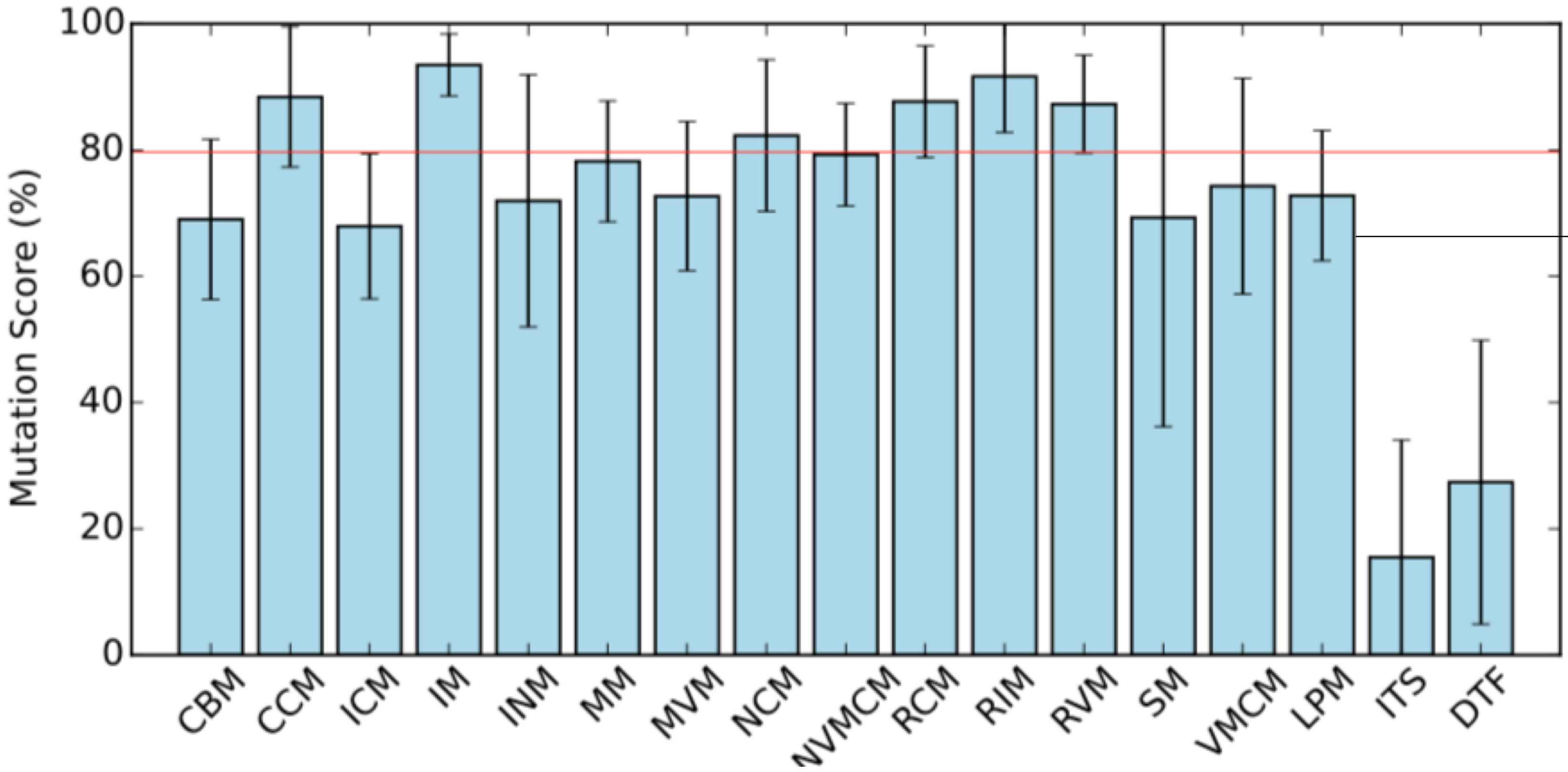
Sufficient Mutation Operators

Mutation Score



79.65%

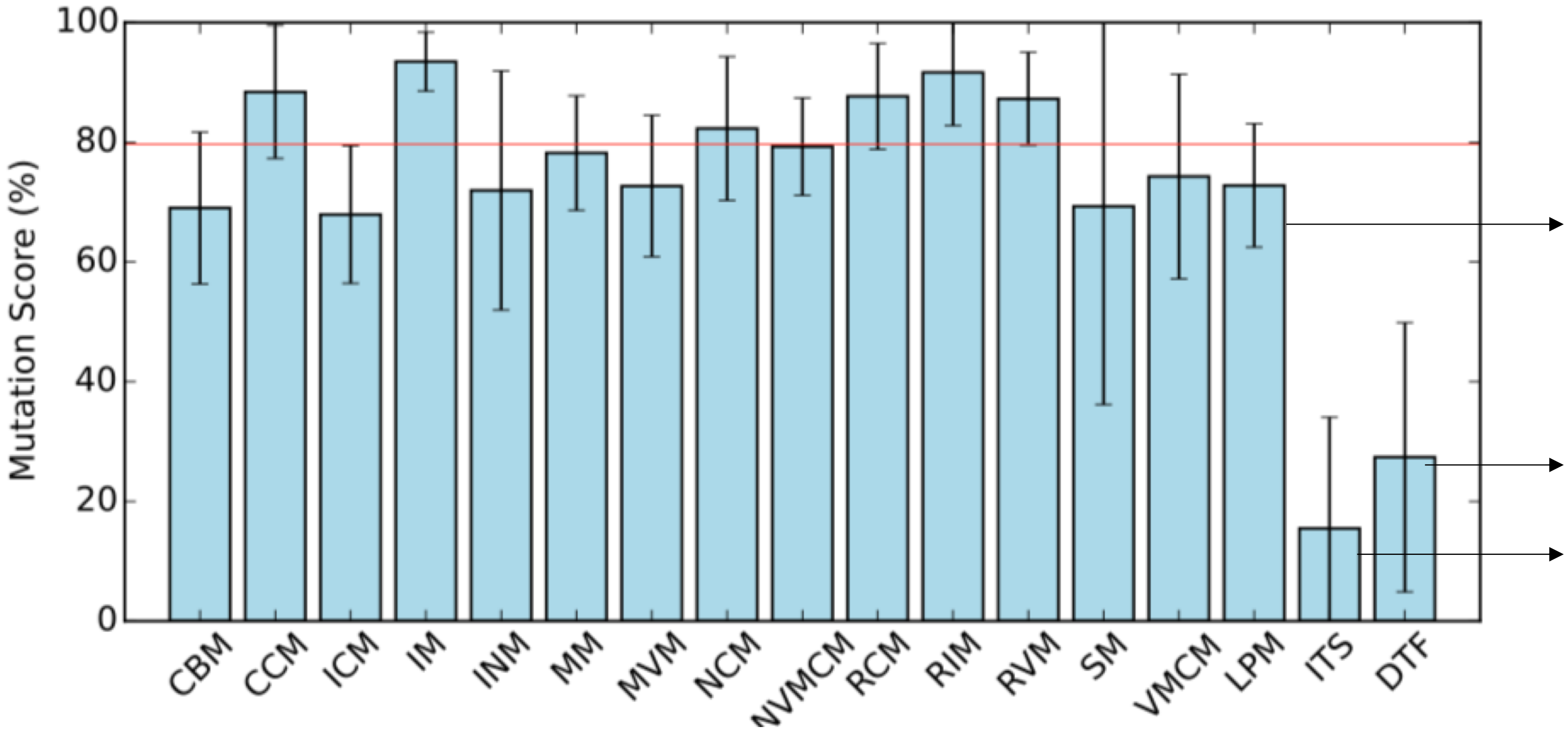
Mutation Score



79.65%

72.78%

Mutation Score



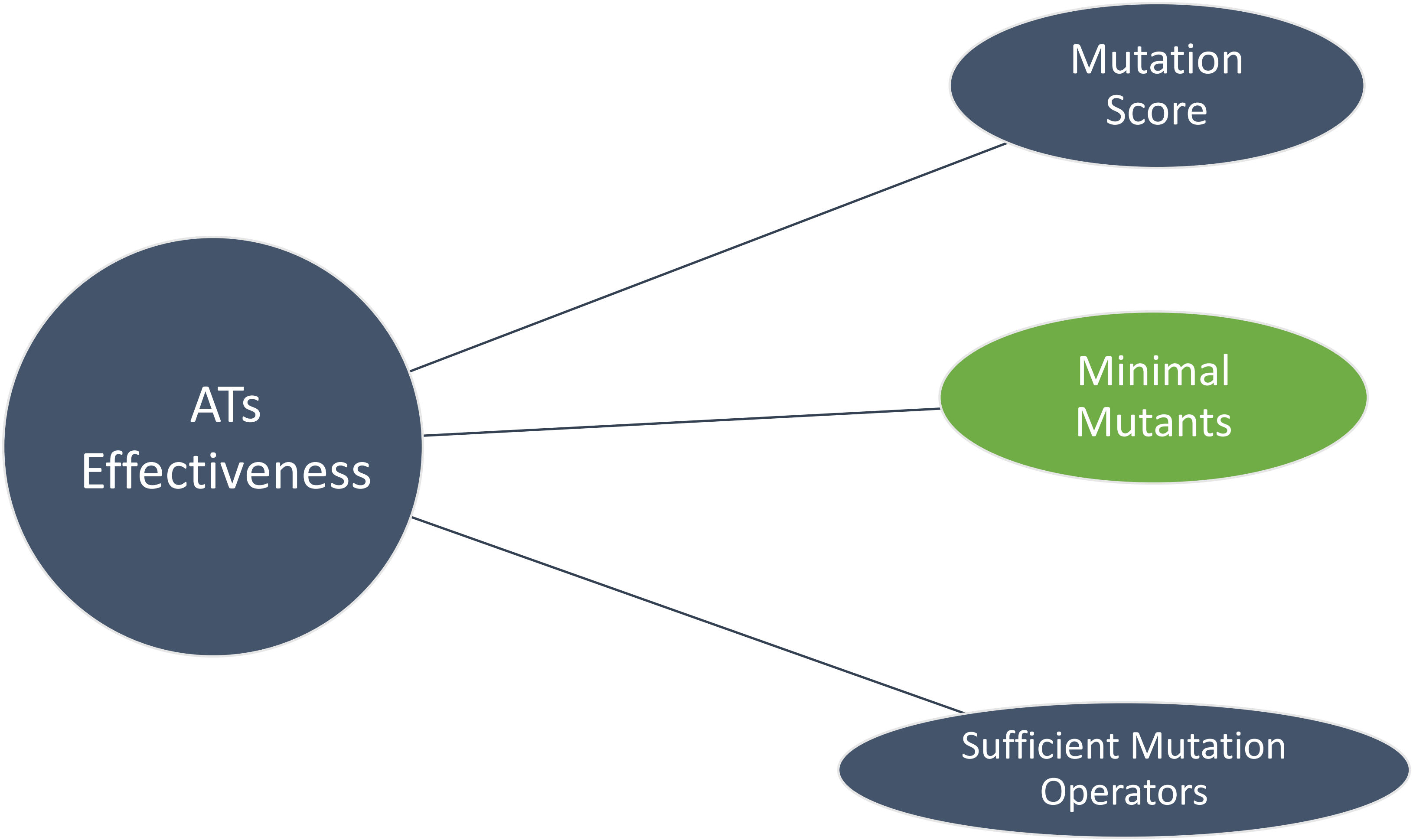
79.65%

72.78%

27.39%

15.49%

Quantitative Analysis



ATs Effectiveness

Mutation Score

Minimal Mutants

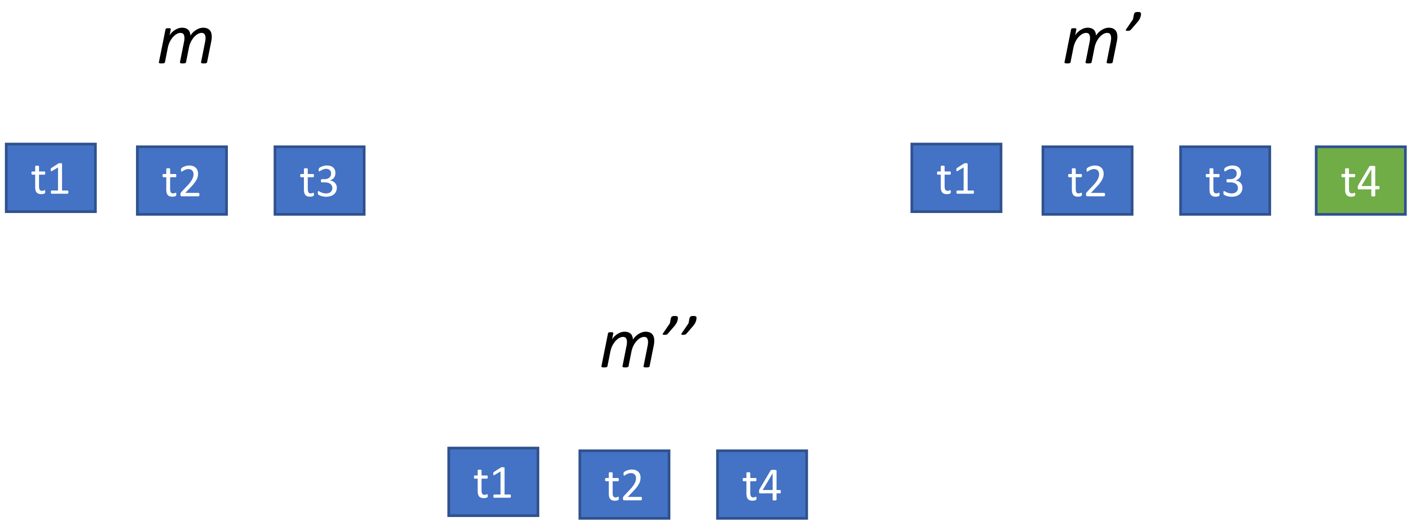
Sufficient Mutation Operators

Minimal Mutants

- Proxies to find the mutants that are harder to kill among the generated ones

Minimal Mutants

- Proxies to find the mutants that are harder to kill among the generated ones
- Dynamic Subsumption:



m

t1 t2 t3

m’

t1 t2 t3 🟩t4

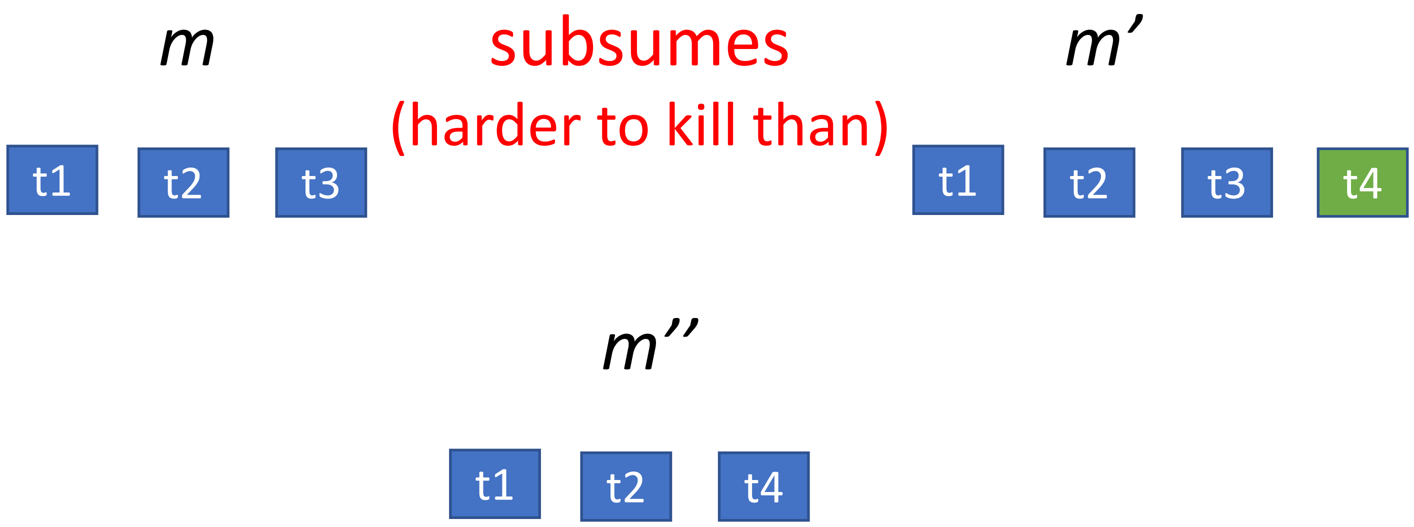
m’’

t1 t2 t4

Minimal Mutants

- Proxies to find the mutants that are harder to kill among the generated ones

- Dynamic Subsumption:



m subsumes (harder to kill than) *m’*

t1 t2 t3   t1 t2 t3 t4

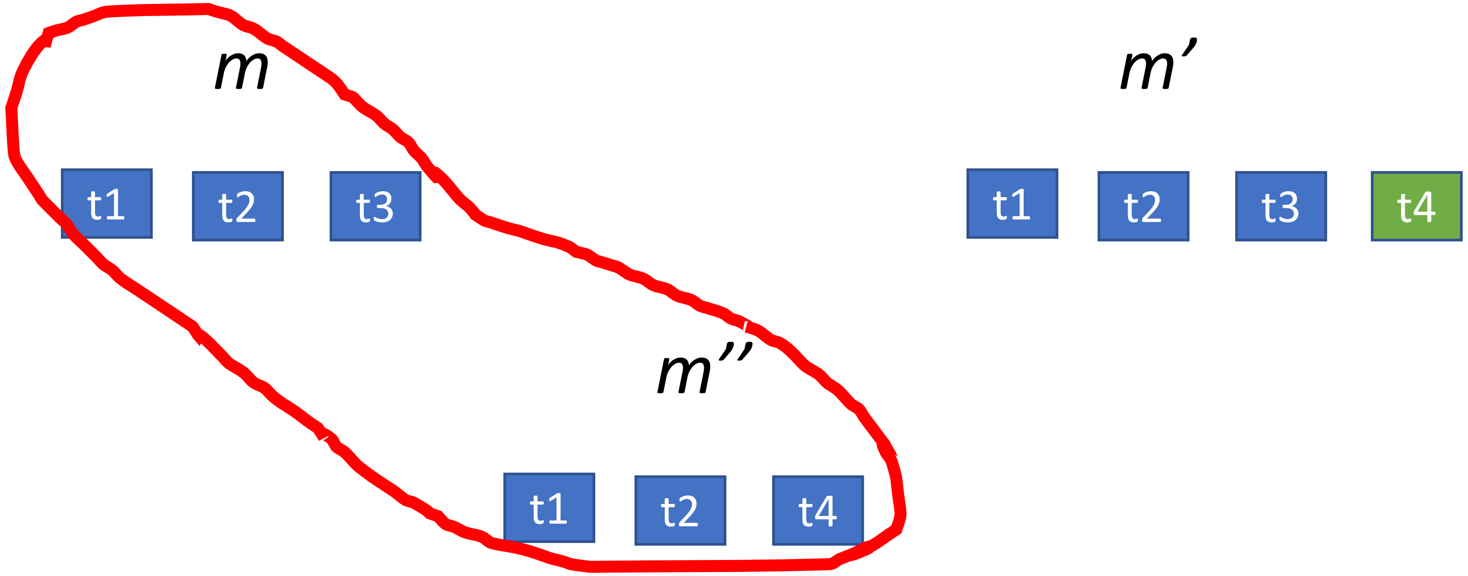
m’’

t1 t2 t4

Minimal Mutants

- Proxies to find the mutants that are harder to kill among the generated ones

- Dynamic Subsumption:



m

t1 t2 t3

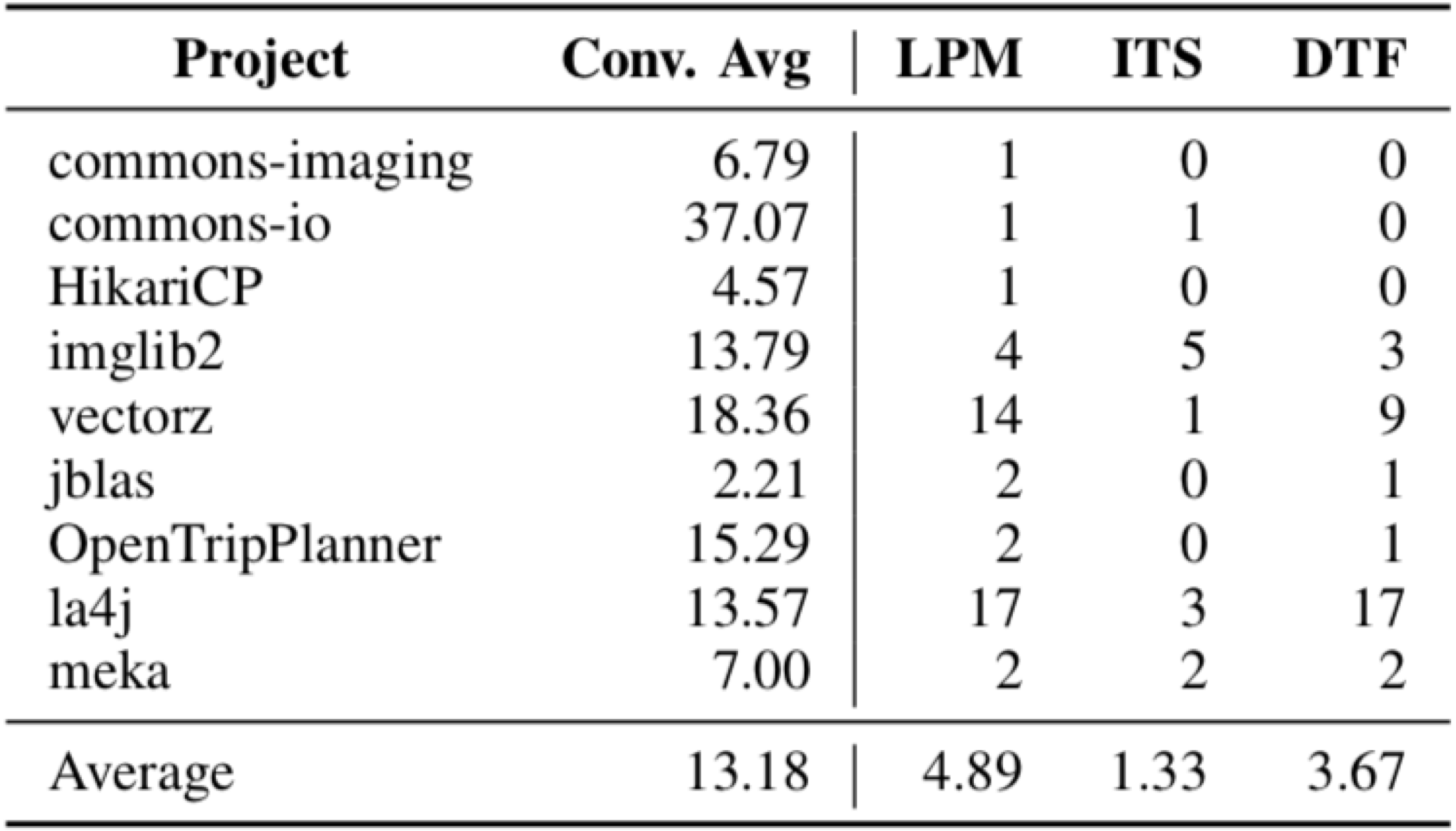
m’

t1 t2 t3 t4

m’’

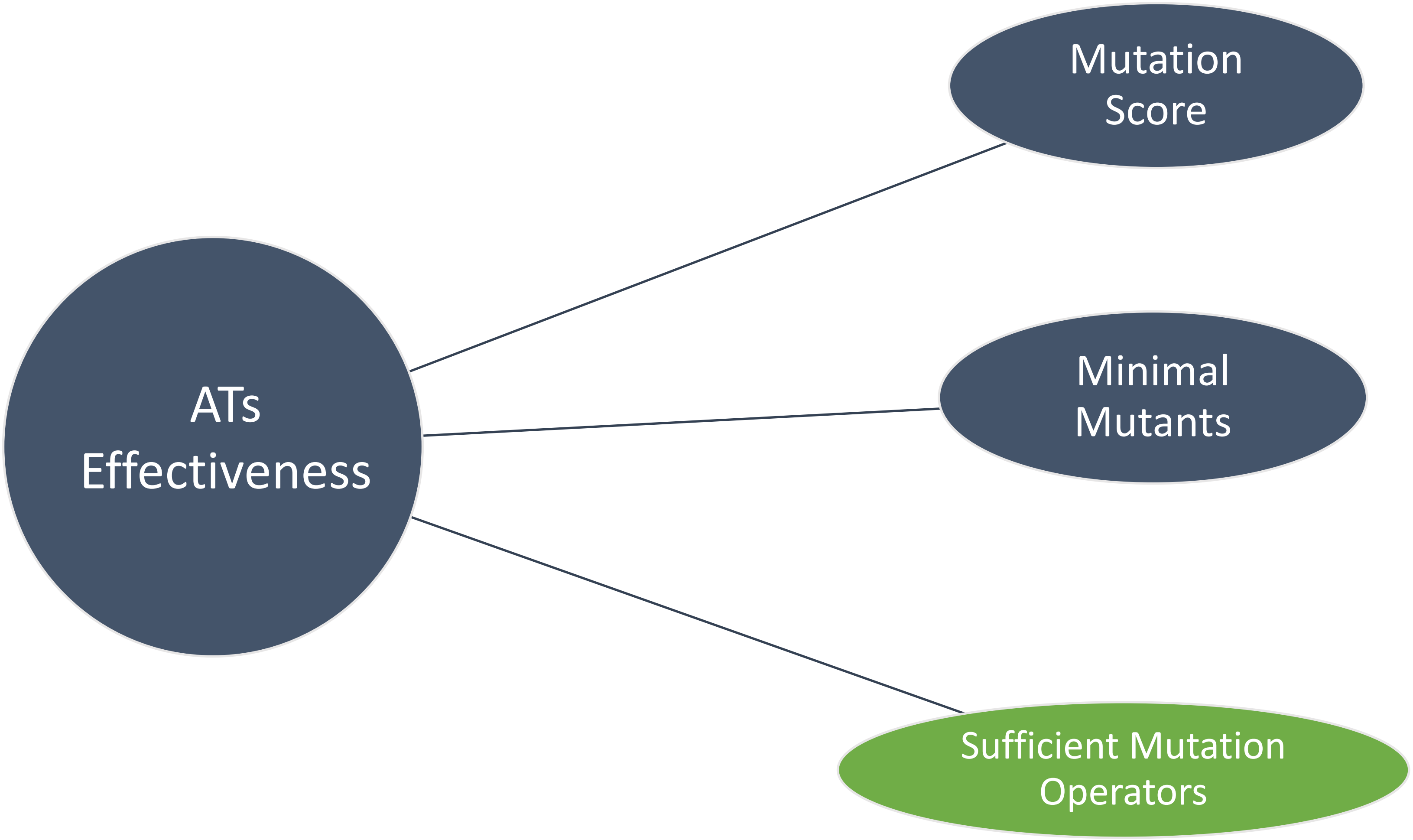
t1 t2 t4

Minimal Mutants



Project	Conv. Avg	LPM	ITS	DTF
commons-imaging	6.79	1	0	0
commons-io	37.07	1	1	0
HikariCP	4.57	1	0	0
imglib2	13.79	4	5	3
vectorz	18.36	14	1	9
jblas	2.21	2	0	1
OpenTripPlanner	15.29	2	0	1
la4j	13.57	17	3	17
meka	7.00	2	2	2
Average	13.18	4.89	1.33	3.67

Quantitative Analysis



ATs Effectiveness

Mutation Score

Minimal Mutants

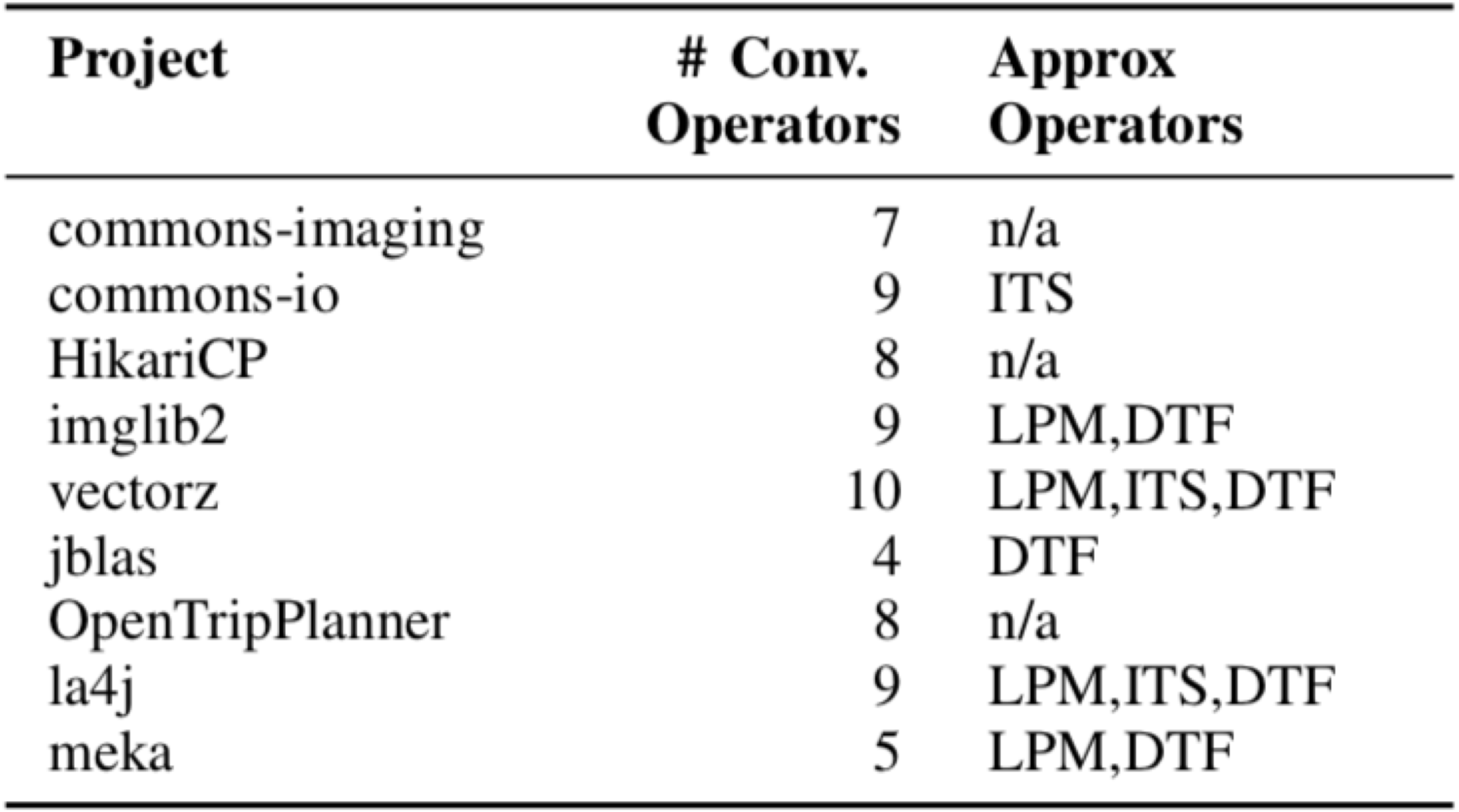
Sufficient Mutation Operators

Sufficient Mutation Operators

- *Selective mutation analysis*: heuristic for reducing the number of mutants to be run

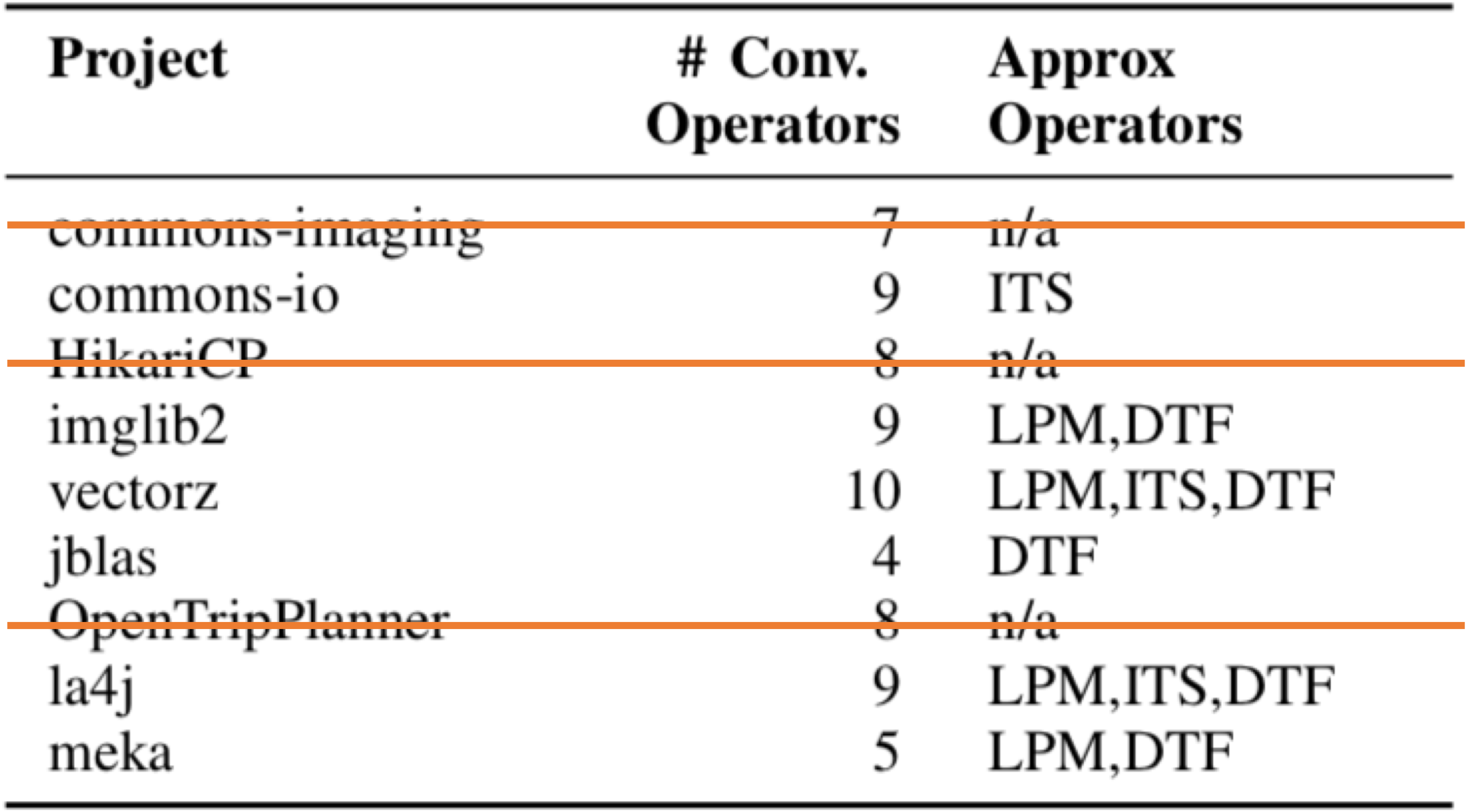
- Tests that kill mutants from the sufficient mutation operators are also sufficient to kill mutants generated by other operators

Sufficient Mutation Operators



Project	# Conv. Operators	Approx Operators
commons-imaging	7	n/a
commons-io	9	ITS
HikariCP	8	n/a
imglib2	9	LPM,DTF
vectorz	10	LPM,ITS,DTF
jblas	4	DTF
OpenTripPlanner	8	n/a
la4j	9	LPM,ITS,DTF
meka	5	LPM,DTF

Sufficient Mutation Operators



Project	# Conv. Operators	Approx Operators
~~commons-imaging~~	~~7~~	~~n/a~~
commons-io	9	ITS
~~HikariCP~~	~~8~~	~~n/a~~
imglib2	9	LPM,DTF
vectorz	10	LPM,ITS,DTF
jblas	4	DTF
~~OpenTripPlanner~~	~~8~~	~~n/a~~
la4j	9	LPM,ITS,DTF
meka	5	LPM,DTF

Summary of RQ1

- LPM mutants are as hard/easy to kill as mutants from conventional operators
- DTF and ITS mutation scores were low due to bad tests
- ATs generate mutants that subsume mutants from other operators
- ATs are a category of their own, not subsumed by other operators
- Results generalize beyond PIT to other mutation tools

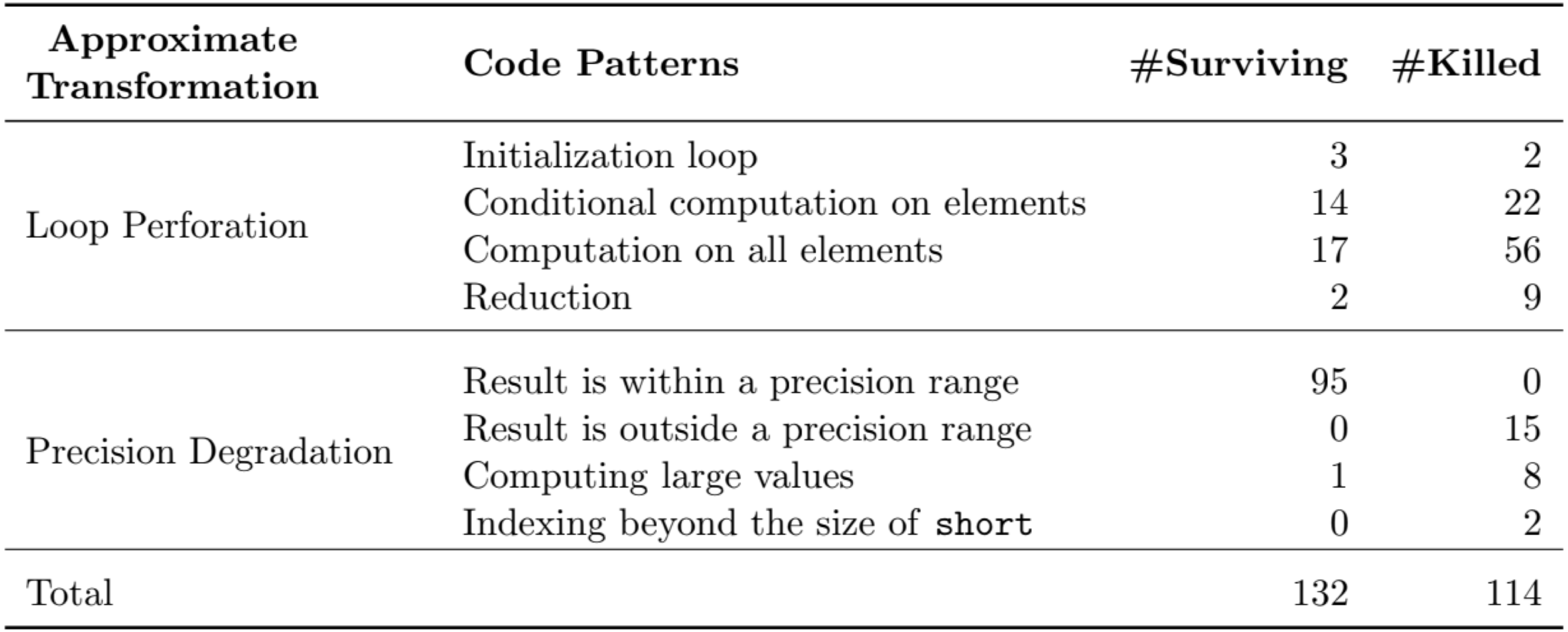
RQ2:

What code patterns do ATs as mutation operators reveal?

Qualitative Analysis

- Sampled 5% of surviving and 5% of killed LPM mutants (125 mutants)
- Sampled 1% of surviving and 1% of killed DTF and ITS mutants (121 mutants)

Code Patterns



Approximate Transformation	**Code Patterns**	**#Surviving**	**#Killed**
Loop Perforation	Initialization loop	3	2
	Conditional computation on elements	14	22
	Computation on all elements	17	56
	Reduction	2	9
Precision Degradation	Result is within a precision range	95	0
	Result is outside a precision range	0	15
	Computing large values	1	8
	Indexing beyond the size of `short`	0	2
Total		132	114

Conditional Computation on Elements Pattern

```
public int argmin() {
 if (isEmpty()) { return −1; }
 double v = Double.POSITIVE_INFINITY;
 int a = −1;
 for (int i = 0; i < length; i+=2) {
  if (!Double.isNaN(get(i)) && get(i) < v)
   v = get(i); a = i;
 }
 return a;
}
```

```
@Test
public void testArgMinMax() {
 A = new DoubleMatrix(4, 3, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0);
 assertEquals(0, A.argmin(), eps);
 …
}
```

RQ3:

How can ATs as mutation operators help software testing practice?

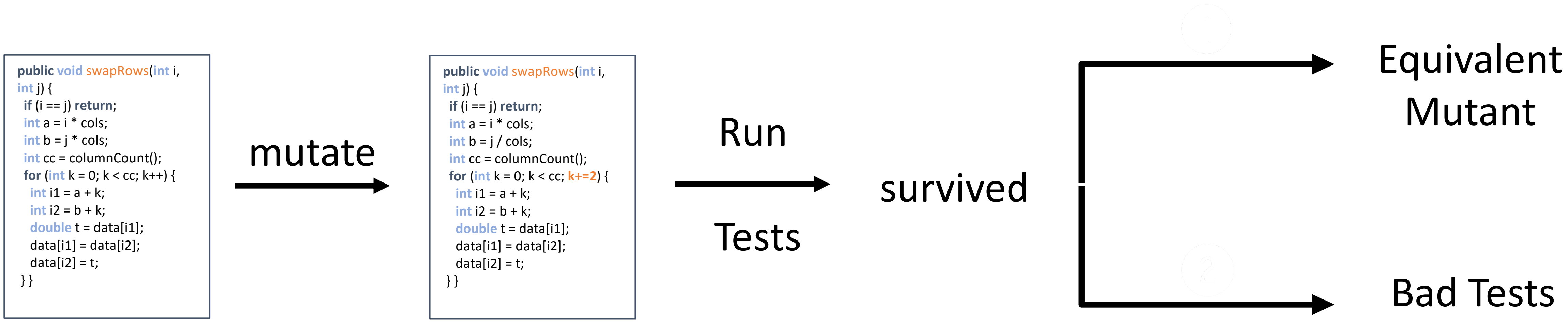
Practical Impact on Software Testing

- Mutation Testing Theory: Interpreting Mutation Testing Results
- Developers Community: Lessons learned, pull requests

Practical Impact on Software Testing

- Mutation Testing Theory: Interpreting Mutation Testing Results
- Developers Community: Lessons learned, pull requests

Interpreting Mutation Testing Results



```
public void swapRows(int i,
int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
} }
```

mutate

```
public void swapRows(int i,
int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j / cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k+=2) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
} }
```

Run

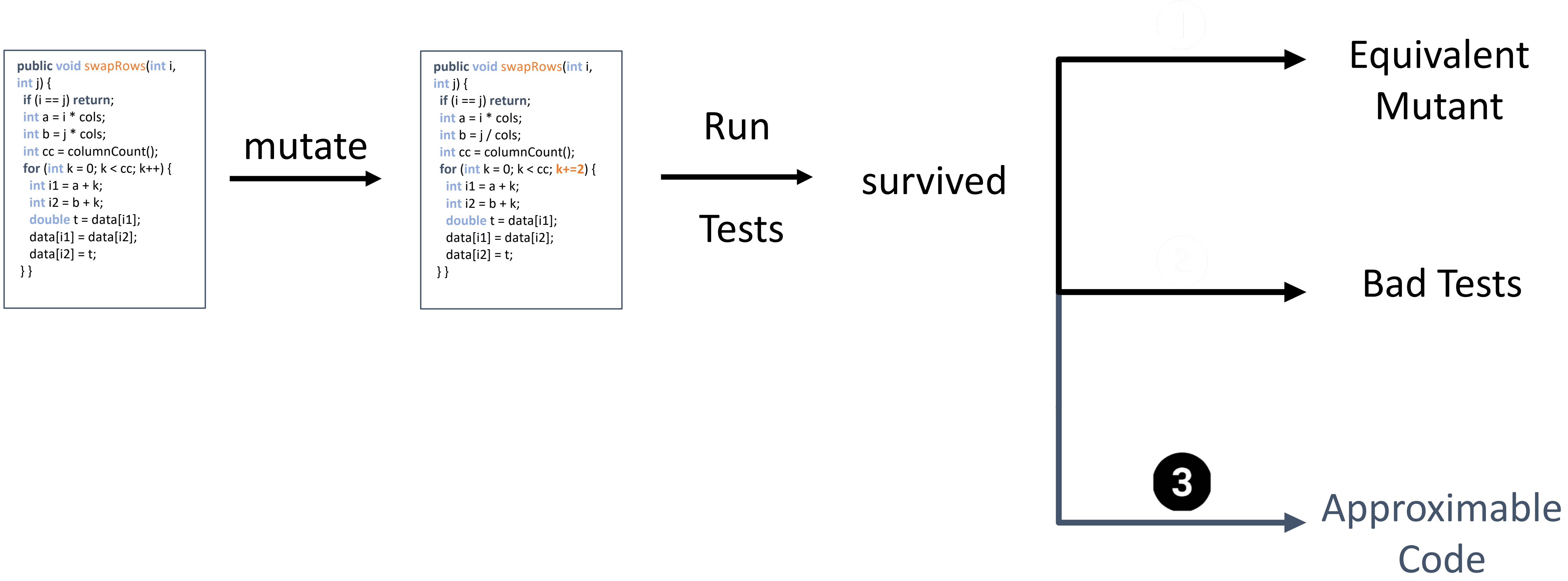
Tests

survived

Equivalent Mutant

Bad Tests

Interpreting Mutation Testing Results



```
public void swapRows(int i,
int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j * cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k++) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 } }
```

mutate

```
public void swapRows(int i,
int j) {
 if (i == j) return;
 int a = i * cols;
 int b = j / cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k+=2) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 } }
```

Run

Tests

survived

Equivalent Mutant

Bad Tests

③

Approximable Code

Interpreting Mutation Testing Results

	Loop Perforation	Precision Degradation
Equivalent Mutants (%)	-	14.58
Bad Tests (%)	63.83	53.13
Approximable Code (%)	19.15	11.46
Hard to Inspect (%)	17.02	20.83

Practical Impact on Software Testing

- Mutation Testing Theory: Interpreting Mutation Testing Results
- Developers Community: Lessons learned, pull requests

Lessons (Re)Learned

1. Better loop coverage
2. Better coverage of loop condition
3. Check all output elements
4. Exercise boundary values

Lessons (Re)Learned

1. Better loop coverage
2. Better coverage of loop condition
3. Check all output elements
4. Exercise boundary values

Bad Test - Pull Request Example

```
public void swapRows(int i, int j)
{
 if (i == j) return;
 int a = i * cols;
 int b = j / cols;
 int cc = columnCount();
 for (int k = 0; k < cc; k+=2) {
  int i1 = a + k;
  int i2 = b + k;
  double t = data[i1];
  data[i1] = data[i2];
  data[i2] = t;
 } }
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2))
return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

Bad Test - Pull Request Example

```
public void swapRows(int i, int j)
{
  if (i == j) return;
  int a = i * cols;
  int b = j / cols;
  int cc = columnCount();
  for (int k = 0; k < cc; k+=2) {
    int i1 = a + k;
    int i2 = b + k;
    double t = data[i1];
    data[i1] = data[i2];
    data[i2] = t;
 } }
```

```
private void doSwapTest(AMatrix m) {
if ((m.rowCount()<2)||(m.columnCount()<2))
return;

m=m.clone();
AMatrix m2=m.clone();
m2.swapRows(0, 1);
assert(!m2.equals(m));

+ assertEquals(m2.getRow(0), m.getRow(1));
+ assertEquals(m2.getRow(1), m.getRow(0));
m2.swapRows(0, 1);
assert(m2.equals(m));
...}
```

Takeaways

- We propose ATs as mutation operators
- Our results show that:
  - ATs generate mutants that are not subsumed by conventional operators
  - ATs exercise unique code patterns
- We propose approximable code as a third way of interpreting surviving mutants
- Survived mutants inspired better testing practices (11 pull requests)